Object file format

- An object file contains the machine code and data for a program, plus other information needed to link and load the program. The machine code and data are stored in three segments:
  - **text**: machine code instructions
  - **data**: initialized data (data that have an initial value)
  - **bss**: uninitialized data (data that have no initial value)

- The segment names are historical. The **bss** segment holds data that is not statically initialized in the program, and is represented in the object file by the amount of memory needed to hold it.
- The other two segments contain information that must be copied from the file into memory before the program runs.
- An object file also contains a symbol table and a patch list. These will be explained below.

---

Slide 20–1

Slide 20–2

Slide 20–3
void assembler(file) {
  dotUF = createFile();
  for line = each line in file do {
    instr = translate(line, Symb); Pc = 0;
    if instr ≠ NULL then {
      syNum = new SymbolTable();
      write(dotUF, instr);
      Pc += sizeOf(instr);
    }
    if hasLabel(line) then {
      syNum = translate(line, Symb); Pc = 0;
      write(dotUF, instr);
      Pc += sizeOf(instr);
    }
  }
}

Assembler—Forward References

<table>
<thead>
<tr>
<th>Loop</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop:</td>
<td>beq $t0, 4, done</td>
</tr>
<tr>
<td></td>
<td>add $t0, $t0, 1</td>
</tr>
<tr>
<td></td>
<td>b loop</td>
</tr>
<tr>
<td>done:</td>
<td></td>
</tr>
</tbody>
</table>

- This program uses symbols (labels). Not knowing the value of done, the assembler can’t translate the bge instruction.
- The assembler must defer translating this instruction until the address of done has been determined.
- There are two ways of solving the problem, both involving first figuring out the symbol addresses, then using them to generate the machine code.

Assembler

The assembler converts one source (assembly code) file into one object file. It performs the following functions:

1. Translate assembly instructions into machine code instructions. This involves setting the bits properly in the machine code.
2. Implement synthetic instructions using one or more machine instructions.
3. Evaluate constant expressions. \( \text{`addu $sp, $sp, 24+8`} \) becomes \( \text{`addu $sp, $sp, 32`} \).
4. Organize data and text in memory. This is difficult because a source file can have multiple data and text segments: the assembler might not know the final address for a label when it is translating an instruction that uses the label.
5. Labels are a form of symbol—a symbolic name for an address.
The first solution is to make two passes over the source file. This is called a two-pass assembler.

**Pass 1:** Compute segment sizes and starting addresses, and fill in the symbol table. The symbol table contains the name of each symbol, and its address. When the assembler sees a symbol defined (such as loop in the first instruction above), it adds it to the symbol table. The first pass size of the segments are left unknown. At the end of the first pass the size of the segments are computed and stored in the symbol table.

**Pass 2:** Translate instructions. If an instruction uses a symbol, find the symbol in the symbol table and use its address during the translation.

- This works, but requires reading the source file twice, which can be slow.

---

### Two-pass – Algorithm

```java
void assembler(file) {
    syTab = passOne(file);
    rewind(file);
    passTwo(file, syTab);
}

SymbolTable passOne (file) {
    syTab = new SymbolTable();
    PC = 0;
    for line = each line in file do {
        if hasLabel(line) then
            syTab.insert(getLabel(line), PC)
        PC += bytesNeeded(line);
    }
    return syTab;
}
```

```java
void passTwo(syTab, file) {
    dotOfFile = createFile();
    PC = 0;
    for line = each line in file do {
        instr = translate(line, syTab, PC);
        if instr != NULL then {
            write(dotOfFile, instr);
            PC += sizeOf(instr);
        }
    }
    close(dotOfFile);
}
```
Another approach is to make one pass over the source file, partially translating the instructions, but make a second pass over the object file to fix (patch) those instructions that use symbols. A patch list is used to keep track of which instructions need to be patched, and which symbol(s) each uses.

**phase 1:** Read entire source file, fill in symbol table, translate instructions (leave unknown addresses blank), and generate patch list. At the end of phase 1 the sizes and starting addresses of the segments are known, and the symbol addresses are computed and stored in the symbol table.

**phase 2:** Process patch list, and patch each instruction using the address of the appropriate symbol(s).

```java
(int PatchList, SymbolTable)
void passOne(file, dotOfFile) {
    syTab = new SymbolTable();
    PC = 0;
    patchList = new List();
    for line = each line in file do {
        if hasLabel(line) then
            syTab.insert(getLabel(line), PC)
        instr = translate(line, SyTab, PC)
        if instr != NULL then {
            if incomplete(line) then {
                patch = new Patch(instr, PC)
                patchList.add(patch);
            }
        }
        write(dotOfFile, instr);
        PC += sizeOf(instr);
    }
    return (patchList, syTab);
}
```
3. Either:
(a) Read the instruction to be patched from the object file, parse it, and patch it depending on what instruction it is, or
(b) Store the type of instruction in the patch list to avoid re-parsing the instruction, or
(c) Store generic patching information in the patch list (e.g. put the address into bits 0-15 of the instruction). This solution doesn’t depend on the instruction encoding or instruction set – it can easily be ported to a different architecture.

Branch instructions

- Branch instructions are PC-relative, they modify the PC by a relative amount, rather than set it to a fixed address.
- This is a good thing, because it means a branch to an instruction in the same segment works even if the segment’s starting address changes (because it’s relative).
- Once PC-relative instructions have been patched, they don’t need to be re-patched if the segment is relocated in memory.

Patch List: Caveats

1. Symbol addresses aren’t known until the end of phase 1. Store each symbol address in the symbol table as an offset from the beginning (base address) of its segment. During phase 2, add the offset to the segment base to get the symbol address.

2. Different instructions have different encodings, which means that patching is instruction-dependent:
   (a) ’lw $t0, foo’ is translated into 0x8C080000 in the first pass. The last four zeros (16 bits) should be patched with the address of foo when the patch list is processed.
   (b) ’jal foo’ is translated into 0x0C000000 in the first pass. The last six zeros plus two bits of the ’C’ (26 bits total) should be patched with the word address of foo. The low 2 bits of foo’s address will always be zero (because instructions must be word-aligned), so leave them off.
When patching instructions the linker must be careful to use the correct symbol table, as several object files might define and use the same symbol. There are two types of symbols:

**Private:** a private symbol is defined and used in a source file. It cannot be referenced by another source file.

**Global:** a global symbol is defined in one source file, but may be referenced by any source file.

**Global definition:** the definition of a global symbol. In MIPS assembly, the `.globl` directive indicates that the symbol is global. Symbols without this directive are private.

**External reference:** a reference to a global symbol that is not defined in the current source file.

The linker combines one or more object (.o) files into a single executable (a.out). To do this it must:

1. Combine like segments. All text segments from the object files are combined into one big text segment, and all data segments are combined into one big data segment.

2. Segment starting addresses will change, so instructions will have to be re-patched. The symbol tables and patch lists stored in the object files are used for this.

3. Resolve external references. If one object file has an undefined reference to a symbol, the linker searches the global symbols defined in the other object files to find a match and patch the instruction. Otherwise an error is reported.

4. Link in object files from specified libraries. A library is a collection of object files that are indexed by the global symbols they define. If an undefined reference is found in the index, the linker extracts the proper object file from the library and links it into the executable.
An Example

- The following example converts an upper-case string to lower-case and prints it. The file main.s has two text segments and two data segments and calls the routine print which is defined in the file print.s.
- The notation hi(symbol) means the upper 16 bits of the symbol's address.
- lo(symbol) means the lower 16 bits.
- xspim doesn't have syntax to express this.

Loader

- The loader is responsible for taking an executable (a.out) and turning it into a running program.
- The a.out was linked assuming that the text segment starts at address 0, but what if there is already a program running at address 0? The new program will clearly have to start at a different address, meaning that many of its instructions have been patched with incorrect addresses and need to be patched again.

Loader...

- The loader using the symbol tables and patch lists to re-patch the instructions. Once this is done, the text and data segments are copied into memory, and the loader starts the program running at the executable's starting address.
Assemble main.s...

- First assemble main.s into main.o. The assembler processes main.s one line at a time, and creates a symbol table and patch list.

### main.s Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>data+0</td>
<td>Private</td>
</tr>
<tr>
<td>main</td>
<td>text+0</td>
<td>Private</td>
</tr>
<tr>
<td>tolower</td>
<td>text+52</td>
<td>Private</td>
</tr>
<tr>
<td>print</td>
<td>undefined</td>
<td>External ref</td>
</tr>
<tr>
<td>offset</td>
<td>data+8</td>
<td>Private</td>
</tr>
<tr>
<td>loop</td>
<td>text+80</td>
<td>Private</td>
</tr>
<tr>
<td>done</td>
<td>text+100</td>
<td>Private</td>
</tr>
</tbody>
</table>

Assemble main.s...

- The value of offset is data+8 because the string string is stored at the beginning of the data segment. string has 6 characters (don’t forget the 0!), but offset must be word-aligned.

- After all lines in main.s have been processed, the size of the data segment is known to be 12 bytes, and the text segment 112 bytes (28 instructions).

### main.s Patch List

<table>
<thead>
<tr>
<th>Instruction address</th>
<th>Symbol used</th>
</tr>
</thead>
<tbody>
<tr>
<td>text+16</td>
<td>string</td>
</tr>
<tr>
<td>text+20</td>
<td>string</td>
</tr>
<tr>
<td>text+24</td>
<td>tolower</td>
</tr>
<tr>
<td>text+32</td>
<td>print</td>
</tr>
<tr>
<td>text+64</td>
<td>offset</td>
</tr>
<tr>
<td>text+68</td>
<td>offset</td>
</tr>
<tr>
<td>text+76*</td>
<td>done</td>
</tr>
<tr>
<td>text+96*</td>
<td>loop</td>
</tr>
</tbody>
</table>

### main.s Segment Info

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base Addr.</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>0x0</td>
<td>0x70 (112)</td>
</tr>
<tr>
<td>Data</td>
<td>0x70</td>
<td>0xc (12)</td>
</tr>
</tbody>
</table>
The instructions tagged with ‘*’ are branch instructions that are patched with the number of instructions difference between the instruction's address and the symbol's address. E.g., to branch backwards one instruction the branch is patched with -1.

A branch can be patched once the branch is known, even if the final addresses aren't. In the case of the bnez loop the instruction can be patched in the first pass because the loop is at offset 80 and the branch instruction at offset 96. The difference is -16 bytes, or -4 instructions.

<table>
<thead>
<tr>
<th>Address</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>0x3C040000</td>
<td>0x3C040000</td>
</tr>
<tr>
<td>0x14</td>
<td>0x3C840000</td>
<td>0x3C840000</td>
</tr>
<tr>
<td>0x18</td>
<td>0x0000000D</td>
<td>0x0000000D</td>
</tr>
<tr>
<td>0x20</td>
<td>0x8C00000C</td>
<td>0x8C00000C</td>
</tr>
<tr>
<td>0x40</td>
<td>0x8C000000</td>
<td>0x8C000000</td>
</tr>
<tr>
<td>0x41</td>
<td>0x10000000</td>
<td>0x11000000</td>
</tr>
<tr>
<td>0x4c</td>
<td>0x15000000</td>
<td>0x15000000</td>
</tr>
<tr>
<td>0x60</td>
<td>0x10000000</td>
<td>0x10000000</td>
</tr>
</tbody>
</table>

Now do the same for print.s, containing the print subroutine.

.data
newline: ascii"\n"
.text
globlprint
print:
1   subu $sp, $sp, 24
2   sw $fp, 0($sp)
3   addu $fp, $sp, 24
4   li $v0, 4
5   syscall
6   lui $a0, hi(newline)
7   addu $a0, $a0, lo(newline)
8   syscall
9   lw $fp, -24($fp)
10  addu $sp, $sp, 24
11  jr $ra
**Linking**

- The assembly process produces two object (.o) files, `main.o` and `print.o`. Each contains a text segment with the machine instructions, data segment, symbol table, and patch list.
- The linker combines the object files into an executable. It opens both files and computes the sizes and starting addresses of the combined addresses.

**Linking...**

- Assume that the `main.s` text segment starts at address 0, followed by the `print.s` text segment, followed by the `main.s` data segment, and finally the `print.s` data segment:

<table>
<thead>
<tr>
<th>Segment</th>
<th>Starting Address</th>
<th>Shorthand</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.s text</td>
<td>0x0</td>
<td>ntext</td>
</tr>
<tr>
<td>print.s text</td>
<td>0x70</td>
<td>ptext</td>
</tr>
<tr>
<td>main.s data</td>
<td>0x9C</td>
<td>mdata</td>
</tr>
<tr>
<td>print.s data</td>
<td>0xA8</td>
<td>pdata</td>
</tr>
</tbody>
</table>

**Assemble Print.s...**

<table>
<thead>
<tr>
<th>print.s Patch List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction address</td>
</tr>
<tr>
<td>text+20</td>
</tr>
<tr>
<td>text+24</td>
</tr>
</tbody>
</table>

**Applying patches**

<table>
<thead>
<tr>
<th>Address</th>
<th>Symbol used</th>
<th>Symbol address</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x14 (text+20)</td>
<td>newline</td>
<td>0x2C (data+0)</td>
<td>high 16 bits</td>
</tr>
<tr>
<td>0x18 (text+24)</td>
<td>newline</td>
<td>0x2C (data+0)</td>
<td>low 16 bits</td>
</tr>
</tbody>
</table>
- Also note that for private symbols, such as `done`, the linker uses the definition from the symbol table for the file it is patching.

<table>
<thead>
<tr>
<th>Address patched</th>
<th>Symbol used</th>
<th>Symbol address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x74 (ptext+20)</td>
<td>newline</td>
<td>0xA8 (pdata+0)</td>
</tr>
<tr>
<td>0x78 (ptext+24)</td>
<td>newline</td>
<td>0xA8 (pdata+0)</td>
</tr>
</tbody>
</table>

- Combining the segments has caused the addresses in three of them to change. I use the shorthand names `mtext`, `mtext`, `mdata` and `pdata` to represent starting addresses of the segments. The patch lists are used to patch the instructions.

- Note that the instruction at address 0x20 is the call to `print` that the assembler couldn’t patch. The linker uses the global definition of `print` in `print.o` to resolve the reference.

- The linker assumes the text segment starts at address 0. If the program is indeed loaded (copied) into memory at address 0, it will run correctly.

- Running more than one program at a time, however, requires loading all but one of them at an address that is not zero. Suppose the program is loaded at address 1000, instead of 0. This means that the segments’ starting addresses are all off by 1000.
• The loader will have to use the patch lists and symbol tables to once again patch instructions to use the new starting addresses. The program will then run correctly when copied into memory at address 1000.

• MacCabe, pp. Chapter 10, 351–381.